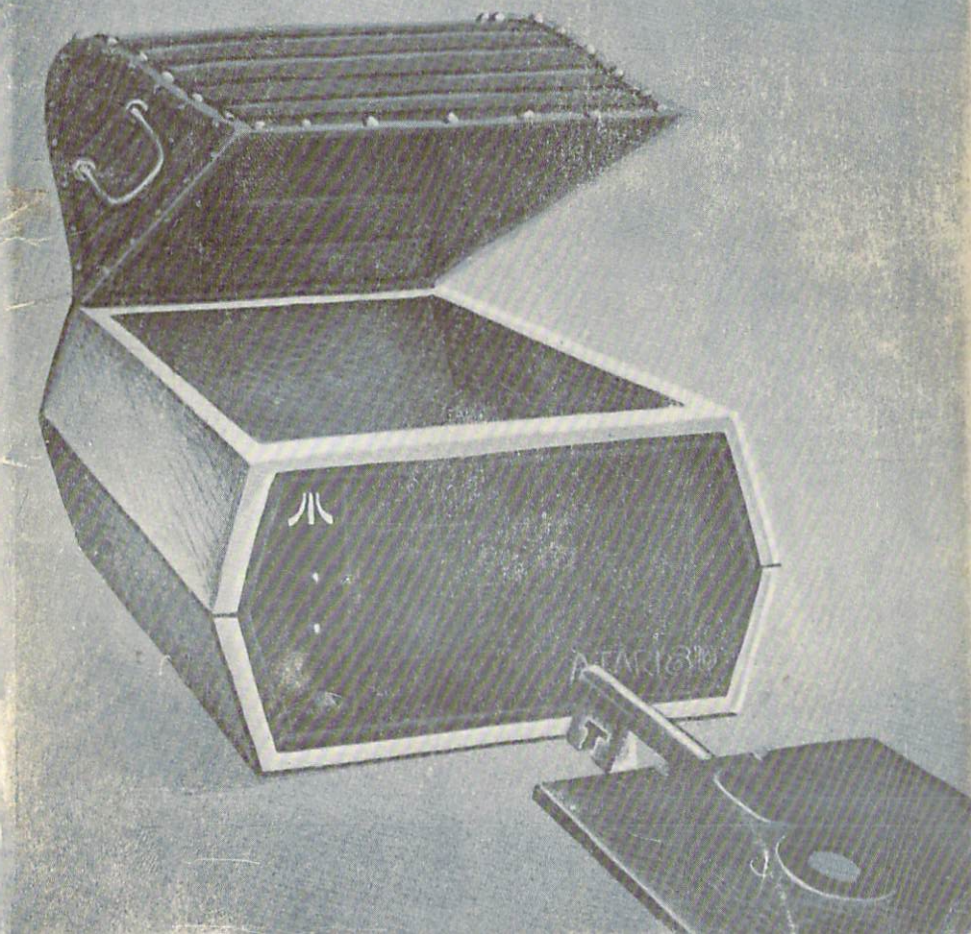
DISKEY

BY SPARKY STARKS



FOREWORD

At the start, I must tell you that this program did not come into being as the result of a carefully-planned effort. The truth is that DISKEY began as the attempt of a desperate programmer to create a few disk utility routines to recover damaged disk information. As often happens, the 'Gee whiz, what else can we make it do?' urge surfaced and DISKEY grew. . . and grew. . . and grew!

I am grateful to the following Atari wizards: Neil Larimer, The Mad Elf, Dan Horn—and especially Scott Adams, Russ Wetmore, and my wife Terri—not only for their suggestions but also for helping me through the darkest hours of 'why won't it work?' despair. Without these collaborators, DISKEY would still be the handful of routines that were its beginning. It would also be slower and more difficult to use.

DISKEY was never intended to be used as a piracy tool. While it does contain powerful automatic copy routines which have the capability to back up almost any material, DISKEY is intended only for legitimate use, which does NOT extend to giving or selling backups of copyrighted programs to your friends [or enemies either]. One's ability to copy someone else's work does not legally or ethically justify theft of that effort. Many companies producing proprietary software have protected their copyright with lucrative reward offers for information concerning even innocent piracy. After considering the damage piracy is inflicting on the computer industry, my sympathy is with the fink plans, not the thieves.

In short, the only right that any user has to the use of any software is the right purchased with that software. If you are using programs that you did not buy from the legitimate source, then you are the thief just discussed. If not, please do not let DISKEY tempt you into becoming that thief.

Lastly, because DISKEY was designed to be an aid rather than a miracle, no claim is made that it will do anything correctly. My best effort has gone into making DISKEY do everything that I [and others] could ask for in the way of disk input/output and related functions. I hope that this program will suit your needs and I thank you for purchasing it.

Sparky Starks

Disk Access and Repair Key

Contents

Introduction

Of the problems encountered in writing software manuals, only one is insurmountable. How do you meet the needs of new programmers without boring the old dogs to tears? DISKEY has been designed to encourage the near-beginner, so some of you had best get out the crying towels. And because I am one of the old dogs myself, there are no doubt concepts that I take for granted, and overlooked as I wrote this manual. Good luck to one and all.

USING THIS MANUAL

The DISKEY manual consists of three main sections. The first is background information, and discusses concepts that you may or may not already be familiar with. It addresses the Atari disk format and *File Management System design philosophy, and discusses some commonly encountered disk problems.

The second section is the start of the operating manual and explains the things that DISKEY does. Suggestions on how to apply DISKEY to specific problems are sprinkled throughout this section.

The third section discusses each functional DISKEY control key and how it is applied. This will serve as your primary reference when using DISKEY. The Table of Contents includes a brief description of each key and serves as a pointer to which key to use for a particular task.

Finally, a glossary is included at the end of the manual to identify both the technical terms that Atari has sanctioned or coined and my own 'wildcat' terminology.

I suggest that you read the entire manual before plugging in the DISKEY disk. Any program that can write to a disk has horrible potential for software demolition. When using DISKEY to modify software, always modify a BACKUP copy when possible. This is very important! Never try to modify an original or *vault copy of software unless you are willing to lose the software. Never, never, never take the write protect off of the DISKEY disk. I destroyed my DISKEY backup three times during development by misusing DISKEY itself. Be very careful.

SPECIAL SYMBOLS

You may have noticed the symbol '*' which appears before certain words in this manual. It indicates that the word can be found in the glossary. If the word is unfamiliar, refer to the glossary for definition before continuing. The '*' symbol appears only with the first use of a word. The '$' symbol is used to indicate a *hexadecimal number and the '%' symbol is used to indicate a *binary number. Symbols D1 to D7 will be used to indicate the *bits of a *byte.

SYSTEM REQUIREMENTS

In order to use DISKEY at all, the following system hardware is required:

1 Atari 400 or 800 computer with at least 32K memory
1 BASIC language cartridge [Atari BASIC]
1 810 disk drive

In addition to the above requirements, DISKEY will be more friendly and powerful if the following optional items are added to the system:

1 additional 810 disk drive
1 printer, 80 column
1 additional 16K memory module [48K total]

DISKEY is a BASIC/Assembly hybrid designed to provide maximum power and flexibility for maintenance and repair of disk based software. The disk on which DISKEY is provided contains several different programs, all of which are required to provide the complete DISKEY command menu. 32K of memory is perfectly adequate for most DISKEY menu options, but systems operating with only one disk drive will require much fewer disk swaps during copy routines if 48K of memory is available for transfer storage.

Section 1. Atari Disk Parameters

Chapter 1. Bytes, Sectors, and Tracks

All eight bit micro-computers store their memory in units called bytes. Each byte equates to a decimal number which is at least 0 and at most 255 [256 different possibilities]. Each of the storage locations in which a byte may be placed is called an address. There are 65536 [256*256] addresses available to the computer because two bytes are used by the machine to number the addresses. Virtually all information used by any computer is processed as bytes stored in memory addresses, so most *peripheral devices [disk drives, for instance] are designed to operate with addresses and bytes. In the Atari single density [standard] disk system each byte is stored in a record called a *sector. Each sector contains 128 bytes of information. Some of this information is data for the computer and some of it is data for the disk system. The disk's sectors are actual physical records stored magnetically in concentric rings on the disk. The rings are called *tracks and there are 40 tracks on a disk. Each track consists of 18 sectors. To recap:

128 bytes per sector
18 sectors per track
40 tracks per disk

This amounts to 128*18*40 or 92160 bytes per disk — more than the entire possible memory in the computer. In addition, the disk can be removed and stored without the constant need for electrical maintenance. What a fine idea!

Actually the 92160 figure is optimistic. It assumes that every byte of every sector of every track is used for data which is useful to your computer and that's not true. The Disk Operating System requires large amounts of the available disk space to keep track of what is where and how the computer will use it. This record keeping system is the source of most disk problems.

Chapter 2. Types of Sectors

The Disk Operating System [usually referred to as the *DOS] is the area of computer code which is assigned to handle a computer's disk drives. Actually, Atari has broken their DOS into two parts. They call these parts the *Disk Handler and the File Management System. The reason for this distinction is that Atari has an *Operating System in the true sense. It is capable of handling most of the special tasks humans request. This includes turning keyboard entries into computer readable bytes, making the monitor screen work, doing all of the sounds and colors, and handling some part of all other input/output functions. One of the input/output routines in the Operating System is the disk handler. From here on, we will call the general part of the Disk Operating system [the part that the computer owns] the *OS. The part that the disk system gives to the computer for use [File Management System] will be called *FMS. When we talk about the whole thing we will use the term DOS.

Normally, the only part of the DOS that is accessibleto the human using an Atari is FMS. FMS in turn relies on the programs in the OS to do its job. DISKEY allows the human to check up on the job that FMS is doing, and fix errors in what FMS has stored on the disk. In addition, DISKEY allows the user to do things that FMS could do if someone had thought of them when designing it. We'll discuss these later.

Right now, lets look at how FMS uses the disk, to help us learn to intelligently play with the information on the disk ourselves.

There are five basic types of sectors written by FMS. The first encountered on the disk is the *Boot Sector. On a DOS II disk, the first three sectors are Boot Sectors. The data in these sectors is written into the computer the minute DOS is activated. They contain information about what FMS looks like, what the disk system looks like [number of drives, etc.], and most importantly, they contain a program to load the computer resident portion of DOS into the computer. This part of DOS is called the Disk File Management system and is contained in the file named DOS/SYS. DOS/SYS must be on the disk in drive 1 when the computer is turned on. It is loaded into memory to support LOAD/SAVE and other BASIC file functions. The DOS/SYS file is always *file number zero.

The sectors occupied by DOS/SYS are a different kind of sector than Boot Sectors; they are called *File Sectors. File sectors contain 128 bytes like any other sector but only 125 contain file data. The last 3 bytes of the sector contain four values that FMS uses to keep tabs on the file. These values are the File Number, [*FN], the number of the next file sector [*NS] [Atari tracks are equated to sector count, giving sector 1-720 or 0-719 depending, but that's another story], the number of bytes in the sector [the last sector of a file is usually a partial record — less than 125 bytes], and a *flag that indicates that the sector is or is not a partial record.

Byte 125 [sector bytes are numbered 0 to 127] contains two things: The first is the FN, contained in D2 to D7. This is used as a check value and is compared with the FN recorded when the file was found in the *directory. If the two FN records do not match, FMS knows that there is a problem. It calls this problem, not surprizingly, FILE # MISMATCH.

Bits D0 to D1 of byte 125 contain the *Most Significant Byte [*MSB] of the *Forward Sector Chain Reference [NS]. This value is multiplied by 256 and added to the contents of byte 126 to find NS. Without this value FMS would not know where to find the next sector of the file.

Byte 127 contains two values. The first is the number of bytes of data contained in the sector. This figure does NOT include the three file control bytes and is stored in D0 to D6 of the byte. D7 bit of byte 127 is a flag. When the flag is on, the sector has LESS than 125 bytes of data. A normal file sector will have a $7D value in byte 127, indicating that the sector is a full sector and that it contains 125 data bytes.

The third sector type is the *Volume Table of Contents sector [*VTOC]. The VTOC is found in sector 360 and is used to show FMS, at a glance, which of the usable file sectors are actually in use. Byte 0 of the VTOC contains a value which indicates with which DOS edition the disk is designed to be used. It will contain 0 for DOS I and 2 for DOS II. Bytes 1 and 2 are the *LSB and MSB of a number that indicates how many file sectors are available for use when the disk contains no files whatsoever. Bytes 3 and 4 are again LSB/MSB and indicate how many free sectors exist on the disk. The next five bytes are currently unused by FMS. Byte 10 begins the VTOC bit map. In each VTOC record byte, every bit

indicates use of one sector. Bytes are read starting with D7 for the lowest sector (backwards from all logical order). A 1 in the bit means that the sector is free while a 0 indicates a sector in use (backwards again). Examination of a blank formatted data disk will show only boot and *directory sectors in use. The VTOC record extends through byte 99 of the sector. The remaining bytes in the sector are not currently used.

The fourth sector type is the Directory Sector; sectors 361 to 368 are so used. Each Directory Sector is divided into eight entries of 16 bytes apiece. The eight sectors contain a total of 64 entries and each entry contains the information used by DOS to find one file. Byte 0 of an entry is a flag byte. D0 of the byte indicates that the file is open (currently in use). This indicates that FMS has damaged the directory if D0 is *set during DISKEY use. This sometimes happens if reboot, system reset, or break are used while writing a record to disk. Never interrupt a disk write procedure at peril of damaging the disk's data. D1 of the flag byte is set to indicate that the file is DOS II format. D3 and D4 are currently unused. D5 is set for locked files. D6 is set to indicate that the file entry is currently being used by a valid file. D7 is set to indicate that the file in this position has been deleted and that this space is available for new directory entries.
Bytes 1 and 2 are LSB/MSB for the total number of file sectors in the file (*T#). Bytes 3 and 4 are LSB/MSB for the number of the file sector of the file. Bytes 5 to 12 are used to store the *ASCII characters of the filename and bytes 13 to 15 store the ASCII filename extension characters. It is easy to see why the length restrictions on filenames and extensions exist.

The fifth sector type is the lost sector. At present, only sector 720 is used for this purposelessness. Somehow a disparity exists between the numbering system used by the computer and that used by the disk drive. The computer thinks that the disk has sectors 0 to 719 and the disk drive numbers the sectors from 1 to 720. As a result, you can legally attempt to read sector zero. FMS won't object, but the drive won't respond. At the other end of the spectrum, FMS doesn't recognize the existance of sector 720 and so does not assign it to files. It is there and DISKEY can access it, but it is not used in normal FMS routines. This has been used to advantage in some software protection schemes.

Chapter 3. Let's Get a File

To better understand how FMS uses its disk organization system, let's follow the file delete procedure. I make no guarantee as to the actual order that the following FMS routine uses, but these are the things that must be done to delete a file.

First, FMS uses the filename that you have provided to find the file in the directory. It attempts to find the file among 8 sectors of 8 file entries each. If no match is found, a FILE NOT FOUND error will occur. If a match is found, FMS will determine that the file is unlocked, not open and not deleted from the first byte in the directory entry. Then, from bytes 1 to 4, the file sector count and first sector will be recorded and the first byte of the directory entry will be modified to indicate that the file is OPEN. Using the First Sector entry, the first sector will be read. The last three bytes of the sector contain the NS and *F# references. The F#reference is compared to the file number implied by the position in the directory in which the filename was found. A mismatch indicates that the sector chain has been damaged and an error will result. If no error is encountered, then the sector is freed in the VTOC and the total sector count is decremented [1 is subtracted]. The NS value is used to read the next record and the process continues as above, with the reading of the sector. FMS uses an NS of zero in the last sector in any file. Error returns will result if a zero NS is discovered before the sector count is done or if, when the sector count is completed, the NS reference is NOT zero. These conditions indicate a damaged file because the file does not agree with the directory.

If no errors have occured, the directory is now updated to indicate that the file is closed and deleted. Note that the file data has not been erased. In fact, the directory information concerning the file should still be intact. The only things that have been changed are the first byte of the directory entry which has been changed to indicate file deletion and the VTOC which has been updated to free all of the sectors formerly used by the file. An accidentally deleted file can be recovered by first modifying the directory to show the file un-deleted and then by fixing the VTOC to re-allocate the file sectors used by the file. Note that if any other files on the disk have been changed since the file deletion, the file sector chain may be in jeopardy. Because the VTOC freed the file's sectors, they may have

since been assigned to other files and rewritten! Disk damage should always be attended to promptly to avoid complications.

When a file is saved, the basic procedure used above changes very little. Instead of following the NS references in existing file sectors to change the VTOC, the VTOC is read to locate free sectors which are then written containing references that agree with the progress of the write process. When a file is loaded, the procedure used for deletion is followed almost exactly. The difference is that in the case of a file load, the sectors are read into memory and the VTOC is not modified. Where the file goes in memory and the type of file being loaded are determined by code imbedded in the file data and are questions best left to the research of the reader. The ATARI [c] Personal Computer System OPERATING SYSTEM User's Manual can answer any questions concerning file data format and most other subjects and is available from ATARI.

Section 2. DISKEY Use and Abuse

Chapter 1. DISKEY'S Screen Variables

In order to make any sense of what DISKEY has to say about a disk, you will need to learn its simple language. In this chapter we will cover most of the variables displayed by DISKEY and explain how each is used. The glossary is a good place to look for anything not found here.

The first variable on the screen is *OS. OS means originate Sector and is used for two things. In manual functions it says from which sector DISKEY is reading. In automatic functions it indicates the starting sector for the function. OS is changed by most routines and should be watched carefully to determine that it is correct before use. OS is changed directly by the [R] and [L] keys.

The next displayed variable is *DS. DS means Destination Sector and indicates the WRITE sector for manual functions and the LAST write sector for automatic functions. DS is set to OS by many routines and should be watched. DS is set directly by [N].

NS means Next Sector and is used to show the forward sector chain reference made by the last sector read. This value is the key to where the file containing it is headed.

F# is short for File Number. It shows in which file the last read sector believes itself to be contained. Actually, the F# value is only correct if the sector is in a valid file chain. Because DOS makes no attempt to erase sectors which have been excluded from files or are in files that have been deleted, the F# value may not be valid. In addition, boot, VTOC, and directory sectors don't use F# or NS and both are meaningless in these sector types.

FILENAME indicates the name of the file selected by the [F],[F] sequence of keys. This is the file on which all file oriented functions will perform.

*OD defines the Originate Drive. In both manual and automatic functions it points to the disk from which you are reading. OD is modified directly by the [O] key.

*DD defines the Destination Drive. It tells you what drive is getting all WRITE instructions. The [D] key toggles the DD variable between drives 1 and 2.

*VE is the Write Verify indicator. If VE says YES, then all writes are re-read to insure correctness. This is the normal state under DOS II. If the VE variable indicates NO!, then no verify is performed. This roughly doubles the speed of all write operations but may lead to errors if your drives and disks are in questionable condition. The VE variable is toggled by the [V] key.

*XR is a little complicated to describe. Some custom files have been found to contain data which has been EOR'ed with some value to be unreadable on the disk. Normally, ASCII data is easily read but a disk EOR confuses ASCII text and makes reading difficult. The EOR value is a binary number to which the data is added, ignoring all Carries. The result of the EOR process is that for each 1 bit in the EOR value, the corresponding bit in the data is switched from ON to OFF or vice versa. For instance, EOR value 128 is %1000 0000 binary. This value, EOR'ed with 100, [%0110 0100], results in data of 228 or %1110 0100. The D7 bit is toggled by the EOR value. Wasn't that fun? Most of you will never use any EOR value but zero, but when you need it, you really need it. Some companies have resorted to using the ROM based character set as an EOR table, mapping each byte in a sector with the corresponding byte in the character table. This results in a hodge-podge that is truly formidable to read. Unfortunately, there is no practical way to include such 'rolling'

EOR values in DISKEY because the minute something is included, it will be avoided in favor of something new by companies trying to protect their software from visual inspection. Despite their disasterous effect on visual inspection, EOR values do not affect duplication techniques. At any rate, good luck with all your EOR's. The XR variable is directly modified with the [X] key.

The T# variable is used with file commands to indicate the Total Sectors in the selected file. It is an aid to determining how far into a file a given sector is and is used with the *S# variable. T# is an implied variable and is not directly modifiable.

The last screen variable is S#. It indicates the relative sector number in a file chosen with the File command. S# is the reference for all file related read and search routines and shows the position in the file when compared with T#. S# is not selectable by the human but is modified by DISKEY during many of the file commands.

Chapter 2. Sector Map, Disk Map

The Sector Map and Disk Map are the heart and soul of DISKEY. These displays show the connected human what is going on inside the disk under DISKEY scrutiny. Actually, the format of each is self-explanatory, but just in case. . .

When a sector is read from a disk, it is placed in an area in the computer's memory called a buffer. It is this *memory buffer that DISKEY represents on the screen as the Sector Map, the contents of the sector under scrutiny.

The Sector Map is divided vertically into two parts: the Hexadecimal Display, and the ASCII display. The hex portion shows the hex value of each of the bytes in the sector and the ASCII part shows the bytes in ASCII. This duality is convenient when searching for text on the right side or code on the left. The power extends a little further in that sectors can be modified in hex or ASCII simply by moving the modify cursor into the appropriate display field.

The two fields are separated by the coarse byte counters. These are added to the fine byte counters under the Sector Map to find a given byte in the sector. Sector bytes are numbered 0 to 127, for a grand total of 128. (Why do these machines think that zero is one?)

The Disk Map shows a record of each sector on the disk, and is used by all multiple sector functions. The meaning of the funny markings varies with map usage, but a general description of symbols is in order.

A period [.] virtually always means that the sector in question was not involved in whatever the Disk Map was used for. A plus [+] usually means that the sector WAS encountered. In file trace, a plus means that the sector is part of a continuous chain; the preceding and following sectors are also in the traced file. A star [*] is used to indicate a starting sector (sometimes called a * renegade]. Starting sectors are not referred to by another sector but ARE included in whatever the function is testing. Inverse characters are used to indicate a forward sector reference that is non-continous; the condition that exists when a file chain jumps over sectors such as when the sector block reserved for the VTOC and directory is encountered. The pound [#] is used to indicate a sector whose forward reference is to sector number zero. These sectors usually indicate the end of a file. A slash [/] is used to indicate a sector that is referred to by a file but by virtue of its file number is assumed to be OUTSIDE of the file. Slashes are a sure indicator of a problem.

The Disk Map display has coarse numbers to the left of the display. To show the whole map, all 40 of the possible screen positions were used and if your monitor has an over-scan problem, part of each coarse number may not show on your screen. For the information of those poor unfortunates, the coarse numbering starts with zero and counts by 36, so the display procedes: 0, 36, 72, 108, 144, 180, 216, 252, 288, 324, 360, 396, 432, 468, 504, 540, 576, 612, 648, 684. The fine sector numbers at the bottom of the screen are added to the coarse numbers to obtain the number of any given sector in the display.

Each line of the map shows two disk tracks so a vertical line is used in the center of the display to separate them.

Chapter 3. What DISKEY Does...Generally

Very little of what follows will make sense without a good understanding of the concepts presented in Section 1. PLEASE read [and reread] Section 1 several times before continuing. If after reading Section 1 you still have difficulty understanding the following chapters, I suggest you read ATARI'S excellent Operating System and Hardware manual before continuing.

This chapter will address DISKEY'S design philosophy. In the course of the remainder of section 2, I will try to explain generally how the program is used to 'operate' on disk problems. Most of the actual key-pushing instruction will be saved for the next section but you should read all that follows carefully to get a feel for where to look in Section 3 to find what you need.

There are two ways to divide DISKEY routines: by what type of sector each is aimed at [file, directory, general], and by what function the routine performs. I will list the command types, and then discuss DISKEY more thoroughly in terms of routine function types. Hopefully, this approach will cover all of the bases with a minimum of confusion.

There are four types of DISKEY keys. The first type consists of keys you just type. These keys perform simple tasks, like changing which drive is being used. Simple keys will be written [X]. The second type is the control group. Control keys are used mostly to perform automated versions of simple keys or to do lengthy or otherwise fancy jobs. I will notate control keys [cX]. The third type is the directory group [!X]. This group allows the directory values of a file to be changed by file number and simplifies what otherwise would be tedious byte interpretation. The fourth key type is the file group [FX]. This group is the most ambitious and allows many of the simple functions to be performed selectively on a previously chosen file.

ROUTINE TYPES

There are eight classifications of DISKEY functions:

1. Read routines
2. Zap routines
3. Informational routines
4. Search routines

5. Error discovery routines
6. Copy routines
7. Repair routines
8. Support routines

These categories are not clearcut (some overlap exists) but they give you some idea of what DISKEY can do. The next chapters will cover each type of routine in detail, and hopefully help to sort out the menu.

Chapter 4. Read Routines

The read group gives you various ways to view a disk's contents. Most read routines do not care about filenames or file boundaries. The OS variable usually defines the drive from which reads are done. The simplest of the routines is (R).

The (R) routine asks you for a sector number (1-720) and then reads the sector and displays it on the sector display. Variables OS and DS are updated to point to the read sector. Remember that the OS variable is used as the lower limit for automatic functions. (R) may be used to set the OS variable for such functions, but (L) is more practical.

The only other keys used purely for reading disks are the relative read keys: they are (+), (-), (F+), and (F-). (+) reads upward from OS on drive OD. (-) reads downward. The file routine versions of these keys read the currently selected file from its first sector to its last. The internal pointers for (F+) and (F-) are preserved from one file trace to the next (see (FT)) so relative file reads may be interspersed with other functions.

All relative read keys will lock on if held until the OS key repeat routine becomes active. To unlock the keys, simply press any non-read key. Relative reads are trapped to the meaningful range of the area read, (1-720 for simple keys, first file sector to last for file keys).

Chapter 5. Zap Routines

Zap routines modify information on a disk as though it resided in memory. Zaps may be compared to the more familiar debug write codes of Assembly Language systems. Most zap functions are simple and quick to use-but beware—ZAP FUNCTIONS DIFFER FROM OTHER ROUTINES IN THAT THEY MODIFY THE SOURCE DISK. This power is necessary for disk repair but has destructive potential. However, all zap routines actually act on an area of memory which has been read from the disk and they offer the opportunity to 'bail out' before re-writing the information to the disk. If you suspect you have done something incorrectly, bail out and start over again.

The simplest zap function is Write. The write [W] routine writes the contents of the buffer to sector DS, drive DD. NOTICE THAT IF DS DIFFERS FROM OS OR DD DIFFERS FROM OD, THE INFORMATION WILL NOT BE RE-WRITTEN TO ITS ORIGIN! Extreme care is required when writing to the disk. [W] and most other zap routines will allow you to place whatever you like on the disk without reference to what can be recognized by DOS later on.

The most general zap function is modify, [M]. The modify routine allows you to replace any byte in the sector display and memory buffer in ASCII or hexadecimal code. On exit from the modify routine you are offered an opportunity to write the buffer to the specified sector on the specified disk. Because the modify function makes no checks to determine the suitability of modifications make sure you know what you are doing. Modify is usually preceded by [R], which gets the target sector into the buffer and insures that DS is set to OS so the sector goes back to where it started on the disk.

The zero [Z] routine offers a quick and dirty way to erase a sector of information. Not actually a zap because no write is offered, this routine quickly clears the display buffer in preparation to write blank sectors to the disk. Note that blank sectors differ from *dead sectors which cannot be read at a later date. Blank sectors still exist but contain the data normally found on blank formatted disks.

The next two zaps, modify forward chain reference [cF] and modify sector file number [cN], are so attuned to repair that they might be better placed in the repair routine chapter. The [cF] and [cN] commands

operate on the sector in the display buffer. They allow you the ability to change the FMS file control parameters of a sector without needing to do boring calculations in binary adjusted integer math. The routines function exactly as their names suggest. Both end with the Sure Response prompt and an offer to write the modified buffer back to the disk.

There is a whole family of file zaps that merely provide functions already available in *XIO form in an easy to use format. As with all file routines, file zaps require the former selection of a target file with the [FF] command. The file zap routines are: [FD], delete file; [FL], lock file; [FR], rename file; and [FU], unlock file. File zaps [FU] and [FL] write to the disk without warning but are reversible. Zaps [FD] and [FR] offer the Sure Prompt before writing the updates to the disk.

The remaining zap routines are the directory [!] functions, all of which qualify as zaps. Each directory entry of a disk lists the following parameters for the entry file:

Filename
Extension
Number of file's first sector
Total number of sectors in file

These attributes are changed by four of the seven directory commands. They are, respectively, [!N], [!E], [!F], and [!T]. None of these routines write anything to the disk. They only modify the display buffer. Changes to the actual disk are made with the directory write [W] sub-menu command. The human can exit the directory sub-menu by using a *null file number entry or with the exit [X] sub-menu command. A new file may be processed with the select file number [!] command. Entry into the directory submenu always requires that you know the NUMBER of the file you wish to update. File numbers may be seen under the main menu directory info [?] command. The directory info command is not available under the directory sub-menu.

DISKEY's zap routines are easy to use, and easy to misuse as well. Always be sure that you are doing what you intended, and that you intended correctly!

Chapter 6. Informational Routines

While most DISKEY routines are informational in one sense, this chapter covers routines that are designed expressly to inform. Routines that qualify as mostly informational but are found elsewhere are: [B], [Q], [S], [R], [cB], [cQ], [cS], [FT], and all of the read routine series.

Of the routines presented in this chapter, the most often used will probably be directory info [?]. This routine begins with the first directory sector [361] and displays each on the screen in a format that is easier to read than the sector itself would be. The information included is each file's file number [0-63], name, extension, first sector, total sector count, and information concerning the status of the file. If the file is locked or deleted, 'L' or 'D' will follow the total sector count. If the entry has never been assigned, 'NOF' will be shown. If the file is DOS I format, an inverse 1 will be shown to the far right. Similarly, if the file has been left in an OPEN state, an exclamation point [!] will appear. The exclamation point is especially useful in that it usually indicates a file that FMS has damaged. Such files are prospects for immediate trace and subsequent repair. In addition to the file information, the [?] command returns the number of the drive queried [OD] and the number of free sectors on the disk. This free sector information is what the VTOC says is free. Addition of this value to the length of all files totaled should give the total space on the disk – 707 for DOS II and 709 for DOS I. If the total is wrong, the VTOC is probably [hopefully] damaged. Otherwise, a file directory entry is incorrect. If the VTOC is incorrect, the [cV] routine will fix the problem while the repair of a file directory entry requires location of the bad file with trace [FT] and continues with experimentation from there. As with most automatic information routines, a [P] key entry will print the screen to a printer if one is available. The [X] key is used to exit the routine; any other key will continue the function one sector at a time until the last directory sector is read.

The print screen to printer [P] routine is available not only as a prompted entry in the directory info routine, but also as a main menu option, as a prompted option in several DISKEY functions, and as an interrupt to most automatic routines. To use the [P] interrupt, hold down the key until the Sure Prompt appears. After the screen is printed, the interrupted function will continue normal operation.

The Disk Map is cleared BEFORE each new use and therefore contains the information gained by a previous use until needed for something new. The print Disk Map [cP] command shows the contents of the Disk Map at any time from the main menu. The [P] command may be used to send a Disk Map to the printer. Note that the label explaining Disk Map use is lost as soon as you leave the routine that generated the map, so you will have to remember how to interpret the Disk Map for yourself.

If you have a printer connected, you can make use of the file memory addresses to printer [FA] routine. This routine works with a binary file that has been selected by the [FF] command and traced with the [FT] command. It re-traces the file, locates the *load block headers, and sends them to the printer in decimal and hex form. By using this routine, you can get a feel for how the code uses memory and what it overlays when loaded. The [FA] routine assumes that the file is BINARY LOAD format and will procede as though it were. If an error results, a FILE NOT PROPERLY ORDERED advice and an error return to the main menu will result. If the printer is not ready a PRINTER NOT READY advice and error return will occur.

A simple but very useful routine is the hex to decimal and ASCII [cH] routine. The routine accepts up to eight hex digits [representing four ASCII characters, etc.] in pairs of two. RETURN is pressed after the last digit to obtain the associated ASCII and decimal data. Note that RETURN is allowed only after even numbered digits and that only valid hex digits may be entered. The exception to this is [X] which may be pressed at any time to abort the routine. the [cH] routine returns the ASCII value of the the first hex digit pair and the decimal equivalent of the entire hex number.

The counterpart to the [cH] routine is the decimal to hex and ASCII [cD] command. This routine requests a number between 0 and 65535. Any entry beyond these bounds or a null entry will get you an ENTRY ERROR return to the main menu. Barring entry error, the equivalent hex value will be returned and if the entry value is less than 256, the ASCII code for the number will also be given.

The last of the informational routines will, no doubt, be the most controversial. Remember, duplication of copyrighted material in any form and for any reason is unlawful. For that reason I cannot suggest that you apply the following technique to any software except that which you have written yourself or for which you have obtained duplication privileges from the copyright holder. On behalf of those who worked hard to produce this and all other copyrighted software, I remind you that it is low-down, mean, and highly unethical to distribute, free or otherwise, any software unless you are under license to do so from the copyright holder.

This routine was designed to allow you to adjust the speed of your disk drives, and it requires that you open the drive unit to adjust it. I DO NOT recommend that you open your drive if it is under warranty. I DO NOT recommend that you open your drive if you are even a little unsure of your technical ability. The speed adjustment procedure is simple and requires only Phillips and standard screwdrivers but will void your drive warranty and is potentially hazardous to the drive. DO NOT USE TOOLS THAT ARE MAGNETIZED. If a tool will pick up an un-attached staple it is magnetized.

First let's discuss what the RPM test [cR] does, and then how it may be used. This routine repeatedly reads one sector of the disk on drive OD and measures the time taken to do so, thereby accurately [+-about .8 RPM, .3%] measuring the rotational speed of the drive. If the speed is in error, the drive will not read data written by a properly adjusted drive. In addition, a drive that is not rotating at the proper speed may be unable to format disks. ATARI drives seem to have a lot of trouble staying at the right speed. Worse, ATARI in its infinite wisdom dictated that the speed of MPI drives should be 285 to 290 RPM, even though they were designed to be used at 300 RPM. Because of this dictum, the strobe disc on the drive mechanism is useless under normal [60 hertz] light.

If you peel off the little stickers on the top of an ATARI drive [the ones that blend in so nicely in each corner], you will find four Phillips head screws underneath. If you loosen each of these screws, the top of the drive case will lift right off. As presently constructed, the drive will not resist this effort—nothing is connected to the lid internally. At the back left corner of the drive, on the back circuit board, lying down flat, is a thumb wheel potentiometer with a screwdriver slot in it. Mine looks like a 5/8 inch flat white button. Turning this button clockwise slows the drive down and turning it counter-clockwise speeds it up.

To adjust the drive, place a good disk you don't mind hurting in the drive and type [cR] under the main DISKEY menu. In about ten seconds, the display will return the drive number tested [OD] and the speed of the drive. Anything between 285 and 290 should be fine. If the drive is slow [less than 285], turn the thumbwheel counter-clockwise a little bit and do [cR] again. Continue this until the drive speed is within bounds. Fast drives need clockwise adjustment of the thumbwheel. DO NOT TOUCH ANYTHING IN THE DRIVE BUT THE THUMBWHEEL, WITH THE SCREWDRIVER OR ANYTHING ELSE! You must have a good disk in the drive to do this routine. [I keep the Phillips head screws in their little pockets in the drive lid, upside-down. Lid removal is thus easily accomplished.]

Now for the controversial part. Some [many?] of the larger software vendors have used damaged sectors on their disks to protect them from duplication. Most verbatim copy routines will *crash when trying to read such sectors. Those that don't crash also don't know which sectors are bad. DISKEY retains a record of dead sectors during its copy routines but there is no standard ATARI function to DESTROY SECTOR. Software vendors generally damage sectors by partially formatting the disk on a foreign [APPLE, TRS-80] system. Such disk areas are totally unreadable on an ATARI machine. Now if a drive is slowed down until it cannot write a given sector, and then slowly sped up until the sector finally writes to the disk without error, the drive usually can't read the sector when re-adjusted to standard speed. The data marks are just too close together at that faster speed to be distinguished. The result of this procedure is that the sector is blown away dead to a normally adjusted ATARI drive. I say usually because I have a newer drive [fast format] in D1 and an older drive with a PERCOM data separator in D2 and D2 can't kill sectors enough to fool D1 . . . but D1 can.

Chapter 7. Search Routines

The search routines are all automatic functions, reading multiple sectors in an organized fashion and with a single goal. There are really only two types of search routine, but one of them is expressed in four routines to cover all of the possible search contexts involved. I'll take the loner routine first, since it is the most confusing.

The file sub-menu offers a routine for finding a location in a file on a disk that would reside in a given place in memory if the file WERE in memory. For instance, if you say, 'Show me the code that goes in memory address $02E3.' the machine says BYTE 122, SECTOR 433 and displays the sector. In addition, the routine will allow you to search for a second place in the file where the same address is again loaded from the file. The reason for the repeat is that for some reason, some software is designed to over-write itself when loading. The routine is called memory equivalent [FM] and requires that the target file be selected [FF] and traced [FT]. This routine uses common code with the [FA] command and gives the same response if the file is damaged or is not a binary load type file. Like all automatic routines [FM] allows interrupt by [P] or [X] keys with the [P] key printing the screen to a printer and the [X] key aborting the routine. The [FM] routine may be very useful in the future to determine which version of a given program you have so as to determine how to apply publisher suggested zaps to faulty code. This procedure is standard practice on many systems and one assumes that it would be adopted by publishers of software for this machine if a way of implementing it were available. Here it is.

The next four commands all attempt to find information on the disk that matches a human supplied key. The first two search from sector 0S to DS on drive 0D. The second two search through a file that has been properly selected and traced. The first and third routines search for a key supplied as hex digits and the second and fourth search for a string of ASCII characters. All allow interruption by the [P] and [X] commands.

The query [Q] routine is used to search for a block of hex digits on the disk on drive 0D. The machine prompts KEY: and you are expected to to enter an even number of hex digits up to twenty, followed by RETURN. The routine may be aborted by pressing [X] during hex entry; otherwise, only valid hex digits are accepted and RETURN is allowed only after even

numbered entries. After key entry, the humam must tell the machine whether it is to use FMS sectors or not. Most searches will be in FMS sectors but autoboot files may be non-FMS. The difference is that FMS sectors reserve the last three bytes of each sector for file handling information and therefore searches that cross sector boundaries must disregard these three bytes — they are not data. The search procedes from sector OS to sector DS and if an exact match of the key is found, the machine displays the appropriate sector and indicates the number of the sector byte at which the key was found. The prompt CONTINUE? is issued to allow further search for the same key. [Y] is the appropriate response if continued search is desired. The [X] and [P] interrupts are allowed.

The search [S] routine is exactly like the query [Q] routine except that an ASCII [text] search key is requested.

The file query [FQ] is exactly like the query [Q] routine except that FMS sectors are assumed and that the search procedes from the first sector to the last sector of a properly selected and traced file.

The file search [FS] is exactly like the query [Q] routine except as listed in both the [FQ] and [S] routines, in other words, the [FS] routine searches a selected file for an ASCII key.

Chapter 8. Error Recovery Routines

Before you get your hopes up, I must warn you that there is no magic wand for fixing disk problems. Error recovery is usually a painstaking process of search and fix. DISKEY's error recovery routines are designed to locate disk problems. After that, it's up to you to make the adjustments on the disk that put it back on track. Error recovery usually requires that you write to the bad disk, so another caution is in order. Don't write unless you know what you are saying! Discussion of error recovery will start with the simple and proceed to the bizarre.

Perhaps the simplest recovery routine is the locate bad sectors [cL] function. This routine scans all sectors on drive OD and prints the Disk Map to show any that defied an attempt to read. The [X] abort key is available in the [cL] routine and can be used to test only part of the disk. If X is used, the resulting Disk Map indicates the last sector read. The

locate bad sector routine does not know if the DATA in sectors is good or bad—only whether or not the actual sectors are damaged. If bad sectors are encountered, they mean that a file chain is probably broken. To determine if this is true you can examine the sectors before and after any dead sector. If the file numbers match and the preceding sector points to the bad one, you can modify the preceding sector to point to the sector following the dead one and reduce the file's total sector count [under the directory menu] to reflect the file length reduction by one sector. This very rarely works but can be helpful in lost Assembly source and sometimes BASIC program files. Some of the file [125 bytes] is lost with this technique but sometimes the file can be pieced back together from the wreckage.

Another dead sector recovery technique is to do a [cC] verbatim copy and then repeatedly read the dead sector until [hopefully] a good read is obtained. Sometimes a sector is marginal and responds only to determined and repeated attempts to read. If the sector can be read even once, the resulting buffer can be written to the appropriate sector of the disk to which the rest of the bad disk was copied. Remember that we are talking about a bad sector, not bad data. If data can be pried from the sector, it can be written to the same numbered sector on an otherwise identical disk and the problem will usually be solved. Dead sectors are a physical problem. They result from damage to the disk surface or a dastardly disk write that was not done according to expected timing parameters. Both circumstances assume that the sector can be read with patience and persistence. Sometimes a power supply interruption will result in a sector that is written too slowly to be read at the normal drive speed. Such a sector can occasionally be read by slowing the drive down for a read of that one sector as elsewhere discussed.

The very simple locate dead sectors routine is only the start of the procedure available for possible recovery of the information lost when the sector died. Another routine used to locate information that may have gone astray is the byte compare [B] routine. Byte compare compares the data on drive D1 to that on D2 in the range of 0S to DS. The Disk Map is used to show which sectors have data that differs. This routine can be used to differentiate between similar versions of autoboot disk software and to search for bit errors on a vault copy of software if the backup is in good shape. Such bit errors can be expected to occur on disks that have been around for awhile.

Before continuing, I would like to offer some suggestions about preventative maintenence. I keep three copies of all software. The most used is the working copy which is in the drive as required. The next is the backup which is reserved for use when disaster destroys the working copy. The third is my vault copy, kept pristine of all custom modifications and safe from the environment. All vault and backup copies are kept in a metal box advertised as a fire vault. The steel is thin but double-walled and may help keep out magnetic bugs. More importantly, the box is nearly vapor-proof and keeps greasy kitchen and cigarette smoke off of the sensitive disks. Similiar boxes are available at department stores and office supply outlets at very reasonable prices, and are recommended to any serious programmer. If, despite all your efforts, a disk comes up with bad data or dead sectors, the triple copy system will prove invaluable when the need for disk repair or recovery occurs.

The control version of byte compare [cB] is identical to the simple version but always searches from sector 1 to 720 inclusive. Both versions allow the [X] and [P] interrupt commands.

The last of the non-repair error recovery routines is the file trace [FT] function. The trace routine operates on a file selected by the [FF] sequence. It is a comprehensive function designed to yield the greatest practical amount of information about any file on a disk without making any more assumptions than are absolutely necessary. It roughly follows the process undertaken by FMS when deleting a file except that the VTOC is not modified. Instead, the Disk Map is used to show the occurance of the file on the disk. In addition, an internal string is used to record the order of occurance of the sectors encountered in the file so that file oriented search and read commands can find the file later without re-tracing. The routine first issues a Sure Response prompt and then searches the directory of disk indicated in the filespec for the file. Trace will always use the first occurance of a filename, even if it points to a deleted version of the file. If this problem is encountered, use the [M] function to change the name of any deleted versions after locating them with the [?] command. The [FR] function does NOT recognize deleted files because it operates through FMS.

Once the file has been found, the trace routine finds the first file sector and uses the file header there to determine what type of file has been found. This is shown on the screen. Assuming that the file is healthy

and normal, the trace function will proceed to the end of the file and then print the Disk Map record of the traced file. The significance of the various symbols used in the Disk Map by file trace are shown immediately below and also discussed in the chapter dedicated to Sector Map and Disk Map.

[.] not involved in file [doesn't have file number]
[+] in file, consecutive sectors before and after
[*] file numbered, not referred to [start sector]
[#] referred to, refers to sector 0 [final sector]
[/] referred to but does not bear file number
[inverse] in file, points to non-consecutive sector

If the file was found intact, this will be shown at the top of the map with the file name and file number. The bottom of this special Disk Map prompts you to exit or locate renegade sectors. Renegades are sectors that believe themselves to be dedicated to a file in use but which are not in the file chain for their file. If a traced file is found to be intact, then any renegades are probably just free sectors that were once assigned to to a file with the number of the file just traced.

Sometimes a file is found that is NOT intact. There are three ways that file trace can find a file that is not intact. The most common results with recourse to the Disk Map and the advice F# MISMATCH, ABS SEC. XX. This advice indicates a blasted file and the Disk Map usually holds the secret to what went wrong. If the Disk Map shows a sector chain of plusses ending with an inverse plus, and a slash [/] in some out-of-the-way place on the disk, the forward sector chain reference of the sector [the inverse plus] which points to that odd-ball spot is probably in error. This is the point where you'll appreciate having a printer. If you do, use the [P] interrupt to print the Disk Map and then select [L] for LOCATE RENEGADE SECTORS. A time-consuming search of the whole disk will result at the end of which a new Disk Map will be printed showing ALL sectors on the disk which contain the file number of the traced file. If the sector immediately following the inverse plussed sector is found to be contained in the file, you are in luck. This condition exists if the first sector following the inverse plussed sector has a star [*] in the second Disk Map [read that again, it does make sense]. If this is the case, modify the inverse plussed sector [that currently points to outer space — the slashed sector] to point to the sector with the star. Now retrace the file and hope it is intact.

If the first Disk Map shows a file of plusses [+] followed immediately by a slash, then the sector with the slash probably belongs in the file [as is claimed by the sector that refers to it] but has been mis-numbered. Again, if you can, send the Disk Map to the printer and then press [L] LOCATE RENEGADE SECTORS. If that slash sector is mis-numbered, it should now be followed by a star [*] or, at least, there should be a star somewhere on the display. Note the sector number of the star sector, or all starred sectors except the one in the first Disk Map, and examine the slashed sector. If it points to one of these starred sectors, it probably belongs to the traced file. Renumber the slashed sector with the file number of the file under examination and try a re-trace. With luck, the file is restored.

Remember the three possible errors encountered in non-intact files? The second and third are FILE TOO LONG, and EARLY EOF. The FILE TOO LONG advice usually means that the directory entry is incorrect but that the file itself is O.K. Under the directory sub-menu, record the existing file length [total sectors] and modify as suggested by the trace routine. Then retrace and actually use the file for whatever purpose it was intended. If no new problems occur, the file is fixed. Otherwise a file update has probably been interrupted somewhere along the line and the file is extremely screwed up. If the latter is the case the file has probably been overwritten here and there and is probably worthless. As a last resort, try restoring the recorded original sector count and pointing the forward sector chain reference from the last file sector [according to the directory] to sector zero. This ploy assumes that that sector is damaged and no longer shows the EOF indicating zero sector reference.

The last trace related error is EARLY EOF. This, is usually the result of the sector showing the EOF forward reference having been somehow zeroed. If examination of the sector with the early EOF shows it to be filled with zero bytes, select the [L] LOCATE RENEGADES procedure. If the sector following the early EOF sector is a start sector [*], try cutting out the bad sector by reference around it or try replacing the bad sector by location of a similar sector on a backup disk. If the sector following the early EOF sector is a [.] sector then you may be in luck. Look for a likely start sector [*] among the renegades to continue. If such a prospect is found, point the sector BEFORE the early EOF sector to the renegade start sector by forward sector chain reference modification.

These procedures are recommended only on the basis of experience. None is guaranteed to work and there is no reason to assume that ingenuity will not suggest better solutions to damaged file problems. These procedures assume that the damaged disk has not been written to since the damage or that the writing has been minimal. Be careful not to damage good files while working on bad ones. If a disk is found to be bad and no backup exists, make a [cC] verbatim copy of the damaged disk before working on it. In this way, if you make matters worse by experimenting, you can at least recover what you started with from the backup disk.

File tracing examines the FMS sector chain procedure, not the data in the file. Files with damaged data require repair that considers the data and not the file. Such repair is available under DISKEY only through such routines as modify [M], and requires considerable knowledge and skill to perform successfully.

Chapter 9. Copy Routines

The copy department is one of the weakest areas of FMS. Normal copy routines always refer to information based on directory information. If the directory could always be assumed to be correct and intact and if all information on a disk could be assumed to be written in FMS file format, everything would be fine, but we all know better. DISKEY's copy routines do not rely on the directory. They are designed to ignore the special nature of each type of sector and treat all as equals. As a result it is possible to copy autoboot disks and disks that have no directory. It is likewise possible to duplicate disks that have been modified to give normal FMS copy routines a tough time. DISKEY does not even assume that the entire disk is copyable. When unreadable sectors are encountered, they are noted in the Disk Map so that duplicate copies can be modified by hand so that they conform to the original – even to the extent of dead sectors. Three copy routines are presented in this chapter. There is a fourth, the special copy routine, that has been placed with the repair routines because it is designed to repair files whose directory references are dead. Such files require the building of new matching directory entries. Of the three routines in this chapter, two are disk to disk copies. They are presented first.

The copy sectors [C] routine is designed to provide a verbatim copy within a sector range supplied by the human. The routine copies all sectors starting with OS and ending with DS from drive OD to drive DD. As with all automatic routines, the [X] and [P] interrupts are honored. Copy functions will destroy the source disk if requested in the wrong direction, so a write protect tab should be placed on the source disk before any copy routine is done... just in case. The Disk Map is used by the [C] routine to record any dead sectors that are encountered on the source disk. All source sectors that are found zeroed [except directory, boot, and VTOC sectors] are ignored during write to save time, so if an EXACT copy is desired, the destination disk should be blank, formatted. [Actually, this should be necessary only for the purist. I have never personally encountered problems with data in sectors that are supposed to be blank.]

The verbatim copy [cC] is identical to the [C] copy except that D1 is assumed to be the source drive, D2 is assumed to be the destination drive, and the copy begins with sector 1 and goes to 720.

The third copy routine is actually a reformatter. It is called tape to disk [T] and is designed to put autoboot tapes on a more convenient format. The routine prompts you to load the tape and reads the first tape record. From this the number of records on the tape is determined and you are requested to rewind the tape and run it again. This time the tape is read and the data is stored in a DISKEY buffer. When the tape is read or the buffer is full, the information from the tape is written to the disk in D1. If the tape proved to be especially long, longer than the buffer, a second rewind and read is performed and the disk is written to again. The tape to disk routine is not universal. Autoboot tapes that have compound structure cannot be written to disk. If the displayed tape record count is much less than the number of records you know to be on a tape, you may assume that the copy will fail. I have run into a few such tapes but these have been the exception. To make room for the large data buffer, tape to disk is a separate routine from the main DISKEY program. For this reason, a wait as the routine is loaded and another as DISKEY is reloaded are normal. The disks created by the tape to disk routine are autoboot disks and totally dominate the disk side on which they reside. UNDER NO CIRCUMSTANCES SHOULD YOU USE ANYTHING BUT A BLANK FORMATTED DISK FOR THE TRANSFER OF AUTOBOOT TAPES. Any information on disks used for this purpose is in jeopardy because the disk

boot record is over-written by the information from the tape. In fact, the record from the tape BECOMES the disk boot record and is treated as such by the disk.

Chapter 10. Repair Routines

Repair routines are useful because they fix problems for you instead of telling you how to do it yourself. Actually there is only one fully automatic repair routine. A second routine is semi-automatic, and two routines are here mostly by default. These two are the erase routines.

The erase disk [E] routine performs a FORMAT of the disk on drive OD. The format is exactly like a DOS format in all respects. The routine is Sure Response prompted.

The pseudo-erase disk [cE] routine performs an update of the boot sectors, the VTOC, and the disk directory without actually re-formatting the disk. If you have an old drive and are dependant on the manufacturer's fast format disks or a friend with a fast format drive, you will appreciate this routine. Pseudo-erase will not change the fast format status of disks on which it is performed. It will also not fix dead sectors, because no format is involved. It WILL generate a disk that performs as though re-formatted in all respects, and is somewhat faster to do than a real format. Pseudo-erase is Sure Response prompted.

The VTOC repair [cV] routine is a complex function that traces each file on the target disk and if all are in order, returns a VTOC record that agrees with the trace process. You can then write this record back to the disk to assure that all sectors in use are reserved properly and that all free sectors are available for use. The VTOC repair routine is suggested when the free and allocated sectors on the disk do not total 707. Note that VTOC repair assumes that the target disk is a DOS II disk. If the disk was generated by DOS I, sectors 2 and 3 will be reserved for boot information that does not exist. This will not cause problems other than that two normally available sectors will be made unavailable. If the disk has been written with DOS/SYS, the loss of the two sectors will not occur... they are reserved for DOS anyway! After VTOC fix of a DOS I disk, the first byte of sector 360 must be changed from two [indicating a DOS II disk] to zero [for DOS I].

If a file problem occurs during the disk trace, an immediate return with the appropriate error message occurs. In this event, no offer to write the new VTOC is tendered. The offending file should be traced with the [FT] function and the problem should be resolved. After the bad file is fixed, VTOC repair can be retried. Entry into VTOC repair is Sure Response prompted and the [X] abort key is available [the [P] interrupt is not].

The most complicated repair routine is special file copy [cS]. The special file copy routine is designed to salvage a file on a disk on which directory sectors are dead. On such a disk, normal copies cannot even begin because FMS fails attempting to read the directory. What the special copy routine does is copy all file sectors from the disk that have the file number of the requested file. Then the Disk Map is printed to show the occurance of the file on the source disk. The special copy routine does not know what sector to use as the start of the file so the human is requested to select one of the start [*] sectors as the first sector of the file. With a little experience, it will become obvious which sector is the likely start of the file; the first start sector is usually the correct start. When the start sector selection has been entered, DISKEY updates the target disk directory, calling the moved file DISKEY/MOV. All sectors encountered during disk read are considered by special copy to be within the file so the file should be traced on the target disk after transfer. If the trace returns a FILE TOO LONG error, the directory entry for the transfered file should be corrected to agree with the number of sectors found during the trace.

Chapter 11. Support Routines

The support routine group is defined as such by default. This section describes the routines that don't fall into any other category. The group breaks down into three areas: routines that set general parameters, routines that perform simple, screen related tasks, and routines that call sub-menus and sub-programs where the real action happens. The support group will be discussed by area.

The select originate drive OD [O], select destination drive DD [D], select new destination sector DS [N], and select function lower limit OS [L] are all self-explanatory. They are used to set parameters for other

routines. The [L] command is distinguished from the [R] routine in that [L] does not read the sector to which OS is set. Generally, OS indicates the last sector read and so is represented in the Sector Map, but after use of the [L] command this is not the case. For this reason, it is a good idea to use [L] only directly before the function for which it is set.

The clear screen [A] routine does just that. It is useful when some advice has managed to hang around and is annoying the human. Please note that the [A] routine also clears any selected filename.

The toggle write verify [V] routine turns on or off the DOS II habit of reading each sector after it is written. If you have drives or disks that are questionable, the write verify should be set to YES. If you trust your system and would like to double the speed of all write operations, set the verify variable VE to NO!.

The upper case only [U] command allows you to convert all lower case characters going to the printer to upper case. Handy if your printer crashes on lower case input.

The EOR value command [X] allows you to read and modify the contents of a disk under a selected bit mask. The mask is useful when data on the disk has had bits toggled to prevent reading. If you don't understand binary arithmetic and the logical 'exclusive or' function, leave the XR variable set to zero. The routine asks for the new EOR value in DECIMAL.

The print current Disk Map [cP] routine shows clears the Sector Map and presents the Disk Map. The map will be set to show its last use. Disk Map is discussed in detail in the trace function section.

The remaining support routines select sub-menus and sub-programs. The first is the [F] command that selects the file sub-menu. After the [F] command, you have the file oriented commands [A], [D], [F], [L], [M], [Q], [R], [S], [T], and [U] to chose from. The file sub-menu is in the main DISKEY program and so no load wait is required. File commands have a priority. All require file selection [FF] to designate the file affected. Most also require file trace [FT] to define the use of the selected file. The file function may be aborted by a null entry for the following file command.

There is one support function in the file sub-menu: select file [FF]. The select file routine allows the human to select the file on which file routines will be performed. When entering the filename, the *Dspec is

optional. If no Dspec is specified, the routine will search both drives for the filename by attempting to unlock the file. Files that are located in this way unlock themselves and should be re-locked if necessary. Because the unlock function is accomplished through XIO, deleted files can't be found and will return a FILE NOT FOUND error. If the Dspec is correctly entered with the filename, [D: D1: D2:], the filename will be accepted without question. This may result in a FILE NOT FOUND error later on in other routines, but has the advantage of selection of a deleted file for file trace. This is the only way that a deleted file can be selected for use of the trace routine.

The select directory sub-menu [!] command does just that. The directory commands then available are [E], [F], [N], [T], [W], and [X]. These functions are all discussed in the chapter on zap routines.

The last function available in the DISKEY menu is the one drive sub-program [cO] command. This command loads an abbreviated version of DISKEY that supports the [R], [N], [C], [cC], and [cS] routines for one drive DISKEY users. The byte compare routines are not available due to the use of the compare buffer space to make more room for sector storage buffer. The memory buffer is adjusted at one drive routine entry to the maximum available according to the memory installed in the system. Return to the main DISKEY program is accomplished by use of the [X] command. This command does not appear on the one drive sub-menu option list.

Section 3. The DISKEY Keyboard

Chapter 1. The Simple Keys

This chapter and the three that follow recap the previous section, in a format designed for reference. Any questions arising from reference here can best be answered in the more complete descriptions in Section 2. The DISKEY directory.

Key: A
Function: Clear screen and filename
Type: Support
Sure Prompt: No
Interrupt: No

The A key performs a clear screen and file variable clean-up. The function results in the selection of NO FILE.

Key: B
Function: Byte compare, D1 to D2, OS to DS
Type: Error recovery
Sure Prompt: Yes
Interrupt: Yes

The B key performs a sector comparison of the disks on drives 1 and 2. The function is normally used to find bit errors on disks for which a valid backup exists, or for routine validation of vault software by comparison with backup.

Key: C
Function: Copy sectors, OD to DD, OS to DS
Type: Copy
Sure Prompt: Yes
Interrupt: Yes

The C key performs a verbatim sector copy within selected parameters. Read and write errors are shown on the screen and all read errors are recorded for on the Disk Map for inspection at the end of the copy function.

Key: D
Function: Toggle DD
Type: Support
Sure Prompt: No
Interrupt: No

The D key selects the alternate destination drive. Note that OD and DD are forced by some functions and should be monitored.

Key: E
Function: Erase disk
Type: Repair
Sure Prompt: Yes
Interrupt: No

The E key formats the disk on the OD drive and then writes sectors 1-4 and 360. Note that no interrupt is provided for in this routine — formatting is handled by the disk drive internal logic and is interruptable only with the break key or system reset.

Key: F
Function: Select file sub-menu
Type: Support
Sure Prompt: No
Interrupt: No

The F key selects the file commands as alternate to main menu keys. All keys discussed in the File Keys chapter require the F key first. Note that [FF] indicates first select file sub-menu and then select filename.

Key: L
Function: Set function lower limit
Type: Support
Sure Prompt: No
Interrupt: No

The L key allows you to arbitrarily set OS as desired. This function DOES NOT perform a disk read and therefore leaves the OS variable at odds with the Sector Map, a condition which does not normally occur and should therefore be noted in this exception. No advice is given to indicate the disparity between OS and the Sector Map so the L function should be used only directly before the automatic functions it defines.

Key: M
Function: Modify
Type: Zap
Sure Prompt: Before exit write offer
Interrupt: No

The M function allows modification of the Sector Map [and the associated memory buffer] directly from the keyboard in ASCII or hex. Exit is implemented by movement of the cursor off of the top or bottom of the Sector Map. The cursor must be BACKED from one Sector Map field to the other. Forward cursor motion in the Sector Map causes wrap-around at the field right margin [cursor drops to start of next line]. Upon exit from the M routine, a Sure Response prompt and write option are offered. The modifications made to the screen are updated on the disk only with this update write operation.

Key: N
Function: New DS
Type: Support
Sure Prompt: No
Interrupt: No

The N key allows you to arbitrarily set the value of the destination sector variable. This variable is used to specify the destination of simple write operations and to specify the last sector affected by automatic functions.

Key: O
Function: Toggle origin drive
Type: Support
Sure Prompt: No
Interrupt: No
The O key selects the alternate source drive by toggling the OD variable.

Key: P
Function: Print screen to printer
Type: Informational
Sure Prompt: Yes
Interrupt: X abort only

The P key operates from the main menu to print the screen display to the printer. All inverse characters are un-inverted before printing.

Dummy characters are substituted for all control characters. In addition to main menu availability, the P control can be used to interrupt any routine for which interrupts are allowed. If used this way, the P routine prints the screen and returns to the interrupted function which then continues. The U command can be used to convert lower case to upper case if your printer does not like the small letters.

Key: Q
Function: Query occurance of hex bytes
Type: Search
Sure Prompt: Yes
Interrupt: Yes

The Q routine allows you to search for a string of bytes expressed as hex numbers. The routine uses the OS, DS and OD variables to define the search area. Offer to continue search follows successful location of search key.

Key: R
Function: Read new OS
Type: Read
Sure Prompt: No
Interrupt: No

The R key allows you to read any sector on disk OD at will. The routine reads the specified sector, updates the Sector Map, and sets OS and DS to the specified read sector value.

Key: S
Function: Search for ASCII string
Type: Search
Sure Prompt: Yes
Interrupt: Yes

The S key performs a string search from sector OS to sector DS on drive OD. The routine conforms in all respects to the Q routine except in that the search key is expressed as an ASCII string.

Key: T
Function: Tape to disk autoboot transfer
Type: Copy
Sure Prompt: Yes, and disk mount prompt
Interrupt: X abort only

The T key loads a sub-program that transfers autoboot tape information to a disk, creating an autoboot disk. See Section 2 for a detailed explanation of this function.

Key: U
Function: Toggle send upper case only to printer
Type: Support
Sure Prompt: No
Interrupt: No

The U key toggles an enable for conversion of all lower case characters sent to the printer to the corresponding upper case characters. The default is no enable (lower case stays lower). If the one drive, special copy, or tape to disk sub-programs are used, the conversion must be re-enabled if used.

Key: V
Function: Toggle write verify enable
Type: Support
Sure Prompt: No
Interrupt: Yes

The V key toggles variable VE which indicates whether every disk write operation is verified immediately by a disk read. The verify procedure is standard in DOS II and is recommended for dependable data transfer; however, disabling the write verify adds 50§ to the speed at which write instructions occur.

Key: W
Function: Write memory buffer to DD
Type: Zap
Sure Prompt: Yes
Interrupt: No

The W key writes the contents of the memory buffer (as reflected on the Sector Map) to sector DS of the disk on drive DD. Be careful to confirm that DS and DD are set as desired before using the W function. Previous contents of the specified disk sector are over-written.

Key: X
Function: Select Sector Map EOR value
Type: Support

Sure Prompt: No
Interrupt: No

A detailed explanation of this function is given in Section 2. Zero is the default and standard value for the XR variable selected by the X key. In addition to the main menu use, the X key is used to abort automatic functions and in this use returns the DISKEY to the main menu.

Key: Z
Function: Zero memory buffer
Type: Zap
Sure Prompt: yes
Interrupt: No

The Z key fills the memory buffer with zero bytes. It is used to provide a clean data field to write to sectors when physically deleting disk information. Remember that the FMS delete function de-allocates the space occupied by the affected file but does not actually erase the file's data.

Key: +
Function: Read upward
Type: Read
Sure Prompt: No
Interrupt: No

The + key reads upward from sector OS on drive OD, updating the Sector Map to show the contents of each sector as it reads. The function will auto-repeat if the + key is held down for one second or more. As a repeating function, + may be cancelled by pressing any key [except +, of course]. On exit from the routine, DS is updated to the value of OS which contains the number of the last sector read.

Key: -
Function: Read downward
Type: Read
Sure Prompt: No
Interrupt: No

The - function is identical in all respects to the + function except that the disk is read downward from the starting OS value.

Key: ?
Function: Show directory information
Type: Informational
Sure Prompt: No
Interrupt: Yes, specially prompted

This key initiates a routine that reads each directory sector and displays all information there in friendly form. File number, name, extension, first sector, total sector count, and status are all given. Deleted, locked, non-existant, left open, and DOS I files are all indicated. More information of this function is available in section 2.

Chapter 2. The Control Keys

Many of the control keys specify versions of the corresponding simple keys, differing only in that the function parameters are predetermined in the control version. Other control functions are special routines that are not commonly used or that use letters that have already been assigned in the simple key menu.

Key: cB
Function: Byte compare, D1, to D2, sector 1 to sector 720
Type: Error recovery
Sure Prompt: Yes
Interrupt: Yes

The cB key compares all sectors of the disks on drives 1 and 2. The Disk Map is used following the operation to indicate any sectors on the two disks that don't match. The cB function is normally used as a preventative maintenance evaluation. Differences in vault and backup or working copies of software serve as an indication of a fault in one or the other. This procedure will discern any variation in disk data without the need to wait until the data error creates real problems.

Key: cC
Function: Verbatim copy, D1 to D2, sector 1 to sector 720
Type: Copy
Sure Prompt: Yes
Interrupt: Yes

The cC key performs an entire disk verbatim copy from drive 1 to drive 2. This type of copy differs from a normal FMS disk duplicate in that every bit of every sector is copied exactly. Any read errors [bad sectors] that are encountered during the cC procedure are shown on the Disk Map displayed at the end of the copy routine.

Key: cD
Function: Decimal to hex conversion
Type: Informational
Sure Prompt: No
Interrupt: No

The cD key allows you to obtain quick conversion of decimal information to hexadecimal and ASCII. The routine accepts whole number decimal entries in the range of 0 to 65535 and returns ASCII value as well for decimal entries in the range of 0 to 255.

Key: cE
Function: Erase disk without new format
Type: Repair
Sure Prompt: Yes
Interrupt: Yes

The cE routine rewrites a disk's boot, VTOC, and directory sectors to conform to those of a DOS II blank, formatted disk. No format [re-write of sector I.D. and timing marks] is performed, so the format type [fast format or early version] is preserved. Dead sectors will remain dead.

Key: cF
Function: Modify sector's forward sector chain reference
Type: Zap
Sure Prompt: At end of routine write option
Interrupt: No

This routine allows the human to modify the DOS file control bytes that indicate the number of the file's following sector. Reference update is in hex. After acceptance of new next sector reference, a write opportunity is presented with the Sure Response prompt. The Sector Map and memory buffer are modified in any event, but the source disk is modified only by election of the write option.

Key: cH
Function: Hex to decimal conversion
Type: Informational
Sure Prompt: No
Interrupt: No

This routine accepts hex data in an entry string of up to eight characters (four digits of hex information). The decimal value of the hex string and the ASCII character corresponding the the first byte of hex are returned by the routine.

Key: cL
Function: Locate dead sectors
Type: Error recovery
Sure Prompt: Yes
Interrupt: X abort only

This routine attempts to read each sector of the disk on drive OD and then prints the Disk Map to show any sectors that could not be read. The routine does not care about the data on the disk; it is confirming the correctness of the sector I.D. and timing marks. X abort is a valid exit for this routine and when used, returns the normal disk map with the last sector read before bail-out indicated on the map.

Key: cN
Function: Modify the sector file number reference
Type: Zap
Sure Prompt: With write option at exit of routine
Interrupt: No

The cN key allows you to arbitrarily change the DOS file control byte which specifies the number of the file in which a sector is contained. The function operates on the Sector Map and associated memory buffer. On exit from the routine, an opportunity to write the updated information to the disk. No disk update is performed unless this option is elected.

Key: cO
Function: One drive sub-program selection
Type: Support
Sure Prompt: Yes, and disk mount prompt
Interrupt: X only, used as normal sub-program return

DISKEY supports most of the functions which normally require two drives for one drive users in a separate program selected by the cO key. A separate program is used is to reserve the maximum possible buffer area for disk information transfer thus reducing the number of disk swaps required. The X key is used internally in the one drive sub-program in the normal manner. In the sub-program main menu, X is used to enable return to the main DISKEY program.

Key: cP
Function: Print current Disk Map to screen
Type: Informational
Sure Prompt: No
Interrupt: Yes

The cP key switches the display from Sector Map to Disk map information. The Disk Map will contain the information recorded by the last routine that used it.

Key: cR
Function: RPM test
Type: Informational
Sure Prompt: Yes
Interrupt: X abort only
This routine returns the rotational speed of drive OD. The drive must contain a formatted disk for the routine to work properly. Section 2 contains more information on the use of this routine for drive speed adjustment and intentional destruction of sectors.

Key: cS
Function: Special copy
Type: Repair
Sure Prompt: Yes
Interrupt: Yes

The special copy routine duplicates all sectors of a selected file number to an alternate disk and then establishes the transferred info as file zero on the new disk. The cS routine is useful where a disk's directory has been damaged and is not recoverable. Considerable judgement and interaction on the part of the human is required. More information is presented in section 2.

Key: cV
Function: VTOC repair
Type: Repair
Sure Prompt: Yes
Interrupt: X abort only

The cV key initiates a routine that traces all of the files of the disk on drive OD and, if all are intact, creates a VTOC record for the disk in the memory buffer. The routine then offers the option to write the new VTOC record to the traced disk. This routine is designed for use with standard DOS I or DOS II disks and should not be used with special purpose or specially protected disks. In such use, the routine will probably damage or destroy the VTOC record for the purposes of the special disk. DOS I disks require that the first byte of sector 360 be set to zero after the routine is finished, [VOTC fix assumes DOS II].

Key: cY
Function: Toggle Sure Response prompt enable
Type: Support
Sure Prompt: Oh, come on!
Interrupt: No

The cY key alternately disables and enables the Sure Response prompt and should be used only with discretion. Disabling of the prompt results in the need for fewer key entries but also inhibits a valuable safety feature. Everyone gets impatient with repetitive key entries, but only the very daring and experienced are qualified to forge ahead with no reminders of their fallability.

Chapter 3. The Directory Keys

The directory keys all function as entries on the directory sub-menu which is selected by the ! main menu key. Unlike the file sub-menu, the directory menu remains active until a return to main menu is specifically requested. Like the file sub-menu, the directory menu operates on a specific file but in the case of directory commands, the file is selected by file number. Since there is no opportunity to request the number of a desired file in the directory menu, know the desired file number when you enter the sub-menu. You may find this number by using the directory info

[?] key in the main DISKEY menu. Unlike the sector modifications associated with file control information, directory information updates DO NOT automatically offer to re-write the affected sector when exiting the modification routine. This option must be selected separately by the directory W command. The W command is effective any time before the directory sub-menu is exited or a new file number is selected that resides on a new directory sector.

Key: !E
Function: Modify directory extension entry
Type: Zap
Sure Prompt: No
Interrupt: No

This routine allows the file extension data of a previously selected directory entry to be changed in the Sector Map and associated memory buffer. No change is made on the disk until the W key is used to re-write the sector on the actual disk record. No check is made by the routine to determine the suitability of changes.

Key: !F
Function: Modify first sector data in directory entry
Type: Zap
Sure Prompt: No
Interrupt: No

The !F routine allows the modification of the record indicating a file's first sector. Update of the disk to agree with the changed Sector Map is not automatic and must be selected with the !W command if desired.

Key: !N
Function: Modify filename data in directory entry
Type: Zap
Sure Prompt: No
Interrupt: No

This routine selects a new filename for a previously-selected directory entry. The filename extension is not changed by this routine but must be altered separately with the !E command. As with all directory modifications, disk update requires use of the !W key and is not automatically offered.

Key: !T
Function: Modify total sector data in directory entry
Type: Zap
Sure Prompt: No
Interrupt: No

This function allows the modification of the sector count for a file previously selected by file number. The routine makes no checks for suitability of changes. This function is normally needed after completion of the special copy routine to correct the normally over-large sector count established by that routine.

Key: !W
Function: Write directory sector to disk
Type: Zap
Sure Prompt: Yes
Interrupt: No

This routine is used to update the disk being zapped in response to changes made by other directory menu operations in the Sector Map. The routine over-writes the previous sector information on the disk from which the Sector Map was read.

Key: !X
Function: Return to main menu
Type: Support
Sure Prompt: No
Interrupt: No

The !X command returns DISKEY to the main menu. The function is required because directory operations return to the directory menu to preserve file number selection until all desired modifications have been made. Therefore, return to main menu is not automatic.

Chapter 4. The File Keys

The file sub-menu is used to specify commands that are oriented to be compatible with the currently used DOS file structure. By use of its commands, files can be dealt with separately instead of as absolute sectors of the disk. The sub-menu supports the expected XIO commands and, in addition, adds some of DISKEY's search and informational

routines for file related use. All file commands with the exception of FF require the prior selection of a file on which to operate. Many of the routines also require previous use of the FT routine to define the boundaries and disk usage of the selected file.

Key: FA
Function: Send binary load file load addresses to printer
Type: Informational
Sure Prompt: Yes
Interrupt: Yes

The FA routine traces the selected file, and, on the assumption that it is a binary load file, sends the load block headers to the printer in binary and hexadecimal notation. The function requires previous filename selection and file trace. If errors are encountered during the header trace, the routine is aborted with a FILE NOT PROPERLY ORDERED error.

Key: FD
Function: Delete file
Type: Zap
Sure Prompt: Yes
Interrupt: No

The FD command deletes the file specified by previous filename assignment. The file to be deleted must be unlocked and available to DOS directory access procedures.

Key: FF
Function: Select filename
Type: Support
Sure Prompt: No
Interrupt: No

The FF routine selects the file on which further file functions are performed. There are two ways that a filename may be specified. If the filename is specified without the use of the Dspec, the routine will search both disks by attempting to unlock the specified file. This procedure will result in a FILE NOT FOUND error if the desired file is deleted or not present on either disk and will result in the selected file being unlocked if found. Alternately, the Dspec can be specified with the filename entry in which case no search for the file is made. This elective will allow the

selection of deleted or non-existant files without challenge; however, a FILE NOT FOUND error may occur later if operations are attempted which require the file to be present and un-deleted. One main advantage of specification of a deleted file is DISKEY's ability to trace and therefore sometimes retrieve a previously deleted file.

Key: FL
Function: Lock file
Type: Zap
Sure Prompt: No
Interrupt: No

This function locks the selected file to normal DOS write and delete access. Notice that file locking does not normally interfere with DISKEY's read/write operations and therefore should not give the DISKEY user undue confidence concerning the security of a locked file.

Key: FM
Function: Memory address occurance in file
Type: Search
Sure Prompt: Yes
Interrupt: Yes

This powerful file routine traces a previously selected binary load file comparing the load block headers with a human provided memory address. If the address is encountered within the file, the byte and sector of occurance are displayed and you are prompted to quit or continue. The FM routine is capable of locating successive occurances of a specific memory address, thus detecting intentional self-overlays within a file. In addition, specific addresses that indicate where the file executes can be located by selection of those addresses as search keys, etc.

Key: FQ
Function: Relative query, find hex key in file
Type: Search
Sure Prompt: Yes
Interrupt: Yes

This routine has been discussed so much that repetition is ridiculous. The relative query differs from simple query in that it searches the sectors of a previously selected file in the order of their occurence in the

file instead of their occurance on the physical disk record. The routine assumes FMS sectors and therefore no FMS prompt is issued. This routine is discussed in detail in Section 2.

Key: FR
Function: Rename file
Type: Zap
Sure Prompt: yes
Interrupt: No

This function uses XIO capability to rename a previously selected file. Unlike the similar directory function, the FR routine renames the file INCLUDING extension. The file must be un-deleted and unlocked. An immediate opportunity to update the disk is ensured after the new filename is selected.

Key: FS
Function: Relative search
Type: Search
Sure Prompt: Yes
Interrupt: Yes

This routine is identical to the relative query [FQ] except that search information is entered as ASCII instead of hex.

Key: FT
Function: Trace selected file, check for inconsistancies
Type: Error recovery
Sure Prompt: Yes
Interrupt: Yes

This powerful and comprehensive routine serves as the basis for many of the other file functions. The routine returns information concerning the type of file, its condition, where it is located on the disk, etc. In addition, these parameters are stored for use by file oriented routines at a later time. If unexpected conditions are discovered in the file, advice is given as to the nature of the problem and extended examination of the file is offered. Further information on file trace and its application is given is section 2.

Key: FU
Function: Unlock file
Type: Zap

Sure Prompt: No
Interrupt: No

This command is used to unlock a previously selected file. The file must be un-deleted and extant to avoid a FILE NOT FOUND error exit.

Key: F+
Function: Read next file consecutive sector
Type: Read
Sure Prompt: No
Interrupt: No

This routine requires previous file selection and trace. It reads forward in the file as the file would be read by DOS in loading, etc. As with the simple + command, the file version will lock on if held for a couple of seconds. Automatic operation after lock will be ended by the pressing of any key.

Key: F-
Function: Read previous file relative sector
Type: Read
Sure Prompt: No
Interrupt: No

This routine is the downward file oriented relative read and corresponds to the + command. The routine is automatically prohibited from exiting the confines of the sectors occupied and will repeat read the first file sector if left unattended while in automatic operation.

Appendix A

Bit/Byte discussion

If you have an understanding of the binary number system, go on to whatever is next. If not, pay close attention! Computer memory consists of a great many on-off switches, called a bits. In the ATARI computer, these switches are arranged in groups of eight. Each group of eight switches are collectively called a memory address or a byte. Your computer is not actually conversant in normal numbers at all. Of the ten (0 to 9) digits in normal or decimal counting, only two can be described with an on-off switch. Computers, therefore, use binary arithmetic. They

count 0, 1, and then run out of digits and have to carry one. The counting thus continues: 0, 1, 10, 11, 100, 101, 110, 111, 1000, and so on. When the computer gets to 11111111 [binary], it runs out of digits in one memory address or byte. Just as each decimal power [1, 10, 100, etc.] represents 10 times the last [10 digits to work with], each binary power or digit represents 2 times the last. The digits in an eight bit memory address represent 1, 2, 4, 8, 16, 32, 64, and 128 if expressed as decimal numbers. If all eight bits are on [1's], we add all of their decimal equivalents and find that the decimal equivalent of the largest number representable in one byte is 255. We rarely need to actually do arithmetic in binary so bits are seldom considered. However, we do run into the confinement of the eight bit memory address or byte, so the 0 to 255 byte restriction is best kept in mind. If you are at all serious about programming, you should be aware that the computer does not store any decimal numbers. All number storage is done in the on-off states of binary arithmetic. A bit is an on-off switch and a byte is eight bits. Your computer stores a byte in each memory address and has 65536 memory addresses it can distinguish. By the way, four bits is called a nybble and there are two nybbles in a byte. Two bits are called a nibble. Isn't this an interesting world?

Appendix B

Hex/Decimal Conversion

If you read Appendix A, or if you already knew about binary arithmetic, you know how incredibly cumbersome it is to use eight digits to express what decimal can say in three. It would be great if there were some simple conversion from decimal to binary to facilitate expression of binary as decimal without the awkward back and forth conversion. There isn't. But there is another available number system that is more compact than binary and DOES convert readily. Enter hexadecimal arithmetic. Hex has 16 digits instead of ten which makes it MORE compact than decimal. Of course, it is almost as alien to decimal as is binary but at least you won't find yourself counting ones, trying to decipher even small numbers. Hex is able to express any eight bit binary number in two digits. To express the extra digits that are not needed in decimal, the letters A to F are used. Here's a hexadecimal count to 10 [or decimal 16]: 0, 1, 2, 3,

4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10. To convert a two digit hexadecimal number to decimal, you merely multiply the value of the high digit times 16 and add the value of the low digit. By this method, hex FF [usually expressed as $FF] is 15*16+15 or 255. Remembering that 255 [decimal] is the largest number containable in a byte, we feel quite smug in the inherent genius of the simple fact that 16 squared is equal to 2 raised to the eighth power, and the whole mess comes out even. Each hex digit in a byte can be called a nybble, giving two nybbles in a byte. Hex is great for expressing binary — better, in fact, than decimal or binary, but how about hex to decimal conversion? The truth is that it's not as tough as binary to decimal conversion but it's still tougher than a trip to the dentist.

Here's a conversion table:

MSN	DECIMAL	LSN	DECIMAL
1	16	1	1
2	32	2	2
3	48	3	3
4	64	4	4
5	80	5	5
6	96	6	6
7	112	7	7
8	128	8	8
9	144	9	9
A	160	A	10
B	176	B	11
C	192	C	12
D	208	D	13
E	224	E	14
F	240	F	15

To convert hex to decimal, simply add one from column A and one from column B.

MSN means most significant nybble and LSN means least etc.

$C3 then is 192 + 3 or 195

Appendix C

Printer Character Conversion

To facilitate use of a number of different printers, the following printer conversion conventions are employed:

All characters above code 127 have 128 subtracted.
All characters below code 32 print as periods [.].

All characters between 124 and 127 print as periods.
Lower case characters are defined by U command.

These conventions disallow inverse and control characters at the printer. [Most printers just make mistakes with such characters anyway.] I have one printer that seems to convert all codes above 128 to line feeds! You should note the characters that your printer makes of codes 91 to 95. These are the characters just above the upper case letters. If your printer does not print lower case characters, you may enable the upper case only option with the U command. U is a toggle which switches the option on and off.

Appendix D

Variable Summary

The screen variables and input conventions are summarized here.

OS Originate sector, read and auto function first sector
DS Destination sector, write and auto function last sector
NS Next sector, DOS forward sector chain reference
F# File number, DOS file number reference in sector
OD Originate drive, drive from which data is read
DD Destination drive, drive to which data is written
VE Write verify status
XR EOR Sector Map print mask
T# File oriented total sector reference
S# File oriented current sector counter

X Commonly available to abort automatic functions
P Commonly available to print screen to printer
Null Entry Common entry abort procedure

Appendix E

Keyboard Summary

A Clear screen and filename
B Byte compare, D1 to D2, OS to DS
C Copy sectors, OD to DD, OS to DS
D Toggle destination drive

E Erase disk (format)
F Select file sub-menu
L Set automatic function lower limit (OS)
M Modify Sector Map
N New destination sector
O Toggle originate drive
P Print screen to printer
Q Query (search for hex key, drive OD, sector OS to DS)
R Read new OS, set DS to match
S Search for ASCII key, drive OD, sector OS to DS
T Tape to disk
U Upper case conversion of printer lower case
V Toggle write verify
W Write memory buffer to sector DS, drive DD
X Select EOR Sector Map screen print mask
Z Zero memory buffer
+ Read upward, next sector on disk
- Read downward
? Directory information
! Select directory sub-menu

cB Byte compare, D1 to D2, whole disk
cC Copy D1 to D2, whole disk
cD Decimal to hex, ASCII conversion
cE Erase disk (without new format)
cF Modify sector forward sector chain reference
cH Hex to decimal, ASCII conversion
cL Locate bad sector on drive OD
cN Modify sector file number reference
cO Select one drive functions sub-program
cP Print current Disk Map
cR RPM test drive OD
cS Special file copy, no directory reference from source
cV VTOC update and repair, drive OD
cY Toggle Sure Response prompt enable

FA File binary load address headers to printer
FD Delete file
FF Select filename for all file functions
FL Lock file

FM Show memory address load position in file
FQ Relative Query
FR Rename file
FS Relative Search
FT Trace file, return file type and file condition
FU Unlock file
FX Return to main menu
F+ File relative upward read, next sector
F- File relative downward read

dE Select new file extension
dF Select new first sector
dN Select new file name, not including extension
dT Select new total sectors
dW Write sector to disk
dX Return to DISKEY main menu

GLOSSARY

ASCII

American Standard Code for Information Exchange. ASCII is the code used universally to express text characters as numbers for transmission between storage and printing devices. For example, in ASCII code, the letter A is a code 65. ASCII also includes codes to signify printer control functions such as line feed and carriage return and supervisory controls (break, etc.). Atari uses an enhanced version of ASCII which includes 256 codes and makes provisions for the Atari control characters; this enhancement is called ATASCII. DISKEY makes no distinction in term usage.

backup

Duplicate, especially for replacement of the original in the event of the original's destruction.

binary

The internal number system used by micro-computers. The binary number system has 2 digits as opposed to the 10 of normal or decimal counting. Binary has only the digits 0 and 1 and therefore lends itself well to the on-off logic of all digital computers. Binary digits are called bits. Binary numbers herein use the % number type marker to distinguish them from other number types.

bit

One digit of a binary number. One of eight digits of a byte.

Boot Sector

A sector on a data disk that is distinguished by the fact that the computer automatically loads its information without any more than internal code and the knowledge that there are disk drives available. Boot sectors do not conform to the standard sector appearance of file sectors and, in fact, are not part of any disk file. DOS II system and data disks begin with 3 boot sectors which are loaded into the computer automatically on power-up if the disk system is detected.

byte

The contents of one of a computer's storage elements. An eight digit binary number. Expressed in the decimal number system, a byte must have a whole number value in the range of 0 to 255. Expressed as hexadecimal, each byte must fall in the range of 00 to FF.

crash

Fail. Bomb. Become blasted. To stop working and resist attempts to re-establish functional status, often with some measure of self-destruction included as insurance.

DD

DISKEY's variable for Destination Drive. The drive to which data is written.

dead sector

A sector that is unreadable by the disk drive. Dead sectors return error messages when read. There are several problems which kill sectors; the data may have been damaged by magnetic fields, the disk surface may have been damaged when touched by almost anything, the drive itself may have damaged the data. A drive that exhibits radical speed changes will often destroy sectors. The Mad Elf has an Atari drive that magnetically destroyed BOTH sides of a disk simultaneously . . . in three seconds! No readable sectors remained, not one. Sector records contain information that is not directly readable on an Atari system: sector timing and I.D. marks that are written when the disk is formatted. One bit error in these marks can kill a sector deader than blazes. People kill sectors too. If the data in a sector is improperly spaced, it may fool the computer into thinking that data is missing.

directory

On a disk, an area reserved to provide the organization for the rest of the disk. The directory names all of the files on the disk and indicates the start, length and status of each.

Directory Sector

One of eight sectors [numbers 361 to 368] that are used to reference the contents of the remainder of the disk as files under the File Management System. Each Directory Sector contains space for eight files and indicates the files' starting sector, total sector count, filename, filename extension, and status.

Disk Handler

Atari controls its devices through a general communication system called CIO. For each device with which CIO can communicate, a routine to identify and specify the device is needed. These routines are called device handlers. The disk handler is such a routine. It is special in the sense that it is not used by CIO directly but rather is controlled by a system called DOS, part of which must be read from a storage disk.

Disk Operating System

A body of code designed to expedite the transfer of information to and from a disk drive storage device.

DOS

Disk Operating System.

DS

DISKEY's variable for destination sector. The sector to which data is written, or, the last sector included in an automatic function.

Dspec

Device specification. In disk files, the drive number reference that precedes the proper filename in the file description.

F#

DISKEY's variable for file number. The internal reference that DOS uses to mark each sector within a file. The position of a filename in the directory. F# has a range of 0 to 63.

File Management System

The disk File Management System, or FMS, or DFM, is the part of Atari's DOS that must be loaded from a disk EACH time is used. The

FMS does the DOS functions that are NOT available in a BASIC environment — file copy, load binary file, etc.

file number

An internal number by which DOS accesses and controls a disk file. Files are numbered according to their occurance in the directory, starting with zero and continuing through the eight directory sectors to file number 63.

File Sector

A disk sector which is in use within a data storage file on the disk. File sectors must conform to a specified data format to be readable by the FMS. This format specifies the use and contents of the last 3 of the 128 sector bytes.

flag

A counter, usually restricted to two possible values, used as a reference to determine the answer to a yes-no question. Alternately, one bit of a byte of information that is read independently of the remainder of the byte to indicate that something is on or off, in condition A or B, etc. DISKEY uses flags to decide if the write verify is on or off, if OD is D1 or D2, if the printer

FMS

File Management System

hexadecimal

A number system that has 16 digits instead of 10, upper digits represented by the letters A-F. Hexadecimal is often used to express binary quantities because of the simplicity of binary to hexadecimal number system conversion and the compactness of hexidecimal numbers in comparison to binary. Hexadecimal numbers herein use the $ number type marker to distinguish them from other number types.

load block header

This is what I call the little numbers that say where binary files are loaded. Binary files have four byte blocks that are taken as two binary integer words [LSM/MSB] and indicate the first and last load addresses of the code which follows the block. A binary file has at least as many such blocks as it has separate areas it wants to load to in memory. This is why a binary file can specify information that is actually written in random

spots all over the computer's memory. Sometimes a file will announce a load block header with a pair of meaningless 255 bytes and sometimes not. Some files use the additional 255's in front of only some of their headers. In any event, the third through sixth bytes of a binary file are always the first load block header. [The first two bytes are ALWAYS 255's, to identify the file type as binary load.].

LSB

Least Significant Byte. One of two bytes used together to extend the number range that the computer can store. See Most Significant Byte for a more detailed explanation.

memory buffer

A buffer is an area where information is stored because it is being transfered between devices with different transfer rates. The buffer is used as an interim place for the information. In the case of Atari disk drives, information is usually needed a byte at a time but cannot be retrieved in other than 128 byte blocks. For this reason, a memory area is reserved into which the 128 byte block is written. The needed byte is then read from this memory buffer. DISKEY's Sector Map is an interpretation of the memory buffer that DISKEY uses for disk data transfer, which is why you must perform a disk write operation after you modify a sector to make the change permanent.

Most Significant Byte

Usually called MSB for short, one of two or several memory address which are used together to specify a single number. Size restrictions of a memory address restrict the contents to the range of 0 to 255, but if a second byte is used exclusively to tally the carry conditions which occur when adding to the number in a memory address, the second address becomes like a second super-digit. Two bytes can thus store 256*256 numbers. These numbers usually have the range of 0 to 65535, but one bit can be reserved to indicate the sign of the number to give half the range as positive integers and half as negative integers. This system is used for a fast computer arithmetic system called integer arithmetic which Atari BASIC does not support.

MSB

Most Significant Byte.

NS

DISKEY's variable name for next sector; the forward sector chain

reference. At the end of every file sector is a code sequence that specifies the next sector of the file. This is the NS.

null

Non-existent. The list of all the Martians reading over your shoulder is a null set [hopefully]. If an entry prompt is requested and you supply an immediate RETURN, the resulting string will have a length of zero and a value of nothing. Null entry is an acceptable way to abort DISKEY routines which call for data entry. The exception to this is hex entry which requires an X entry to abort.

OD

DISKEY's variable for Originate Drive — the drive from which data is read. Also, a dosage beyond the recommended intake level, or the act of intake of such dosage, or the effect of such dosage. Ex: The disk OD'd on CocaCola.

Operating System

A body of computer code that controls the operation of the computer itself, not the jobs that the computer does for you. Keyboard deciphering, screen control, and timing functions are examples of operating system operations. The Atari is a rarity amoung microcomputers because it boasts a real live operating system.

OS

Operating system. Also DISKEY's variable for Originate Sector — the sector from which disk data is read and the first sector in automatic operations.

peripheral

At the edge. In computers, attached to the main unit or associated with it but not contained within. The monitor, keyboard, disk drives, and tape deck are all peripheral devices. Note that the keyboard is a peripheral device because of the way that it is seen by the main computer, not according to its physical location.

renegade

A term I use to indicate a sector that contains data but is not included in any file chain. A renegade does have an assigned file number and may be an estranged sector of a damaged file or just an old sector no longer included in the file to which it originally belonged.

S#

DISKEY's variable for current relative sector number. It refers to a sector by its position in its file. The absolute version of the S# variable is OS, which usually indicates the absolute [disk's] sector just read.

sector

A physical record on a disk drive, so named because it occupies a section of a physical ring or track on the disk surface. Each sector on an ATARI single density disk contains 128 bytes of computer-readable information and is one of 18 sectors on each disk track.

set

On. Yes. Especially as a flag bit that is a one rather than a zero. Enabled as opposed to disabled. Active, initialized.

T#

Total number. DISKEY's variable for total number of sectors in a file.

track

One of 40 concentric rings, each containing 18 sectors of an Atari data storage disk. Atari does not actually use the track concept in its data retrieval scheme, prefering rather to number the sectors 0 to 719 [or 1 to 720, depending], as though they were continous.

VE

The label for the verify flag indicator. YES indicates that a read operation is automatically done after each write operation to ensure data accuracy. This is the normal DOS state. The NO! indication states that no verify of write operations is being performed. This increases disk write speed by 50%.

vault

A copy of software that is not used except when all other copies are unusable, and then only for the creation of duplicates. The 'last chance' replacement, often the original original.

Volume Table of Contents

Usually shortened to VTOC, a sector [number 360] on each data disk reserved to indicate which of the remaining sectors are free for use and which are occupied. In the VTOC, each sector on the disk is indicated as in use if an associated VTOC bit is OFF [or zero]. The Atari VTOC also stores values indicating which DOS the disk was created with, how many free sectors the disk started with, and how many remain free for use.

VTOC

Volume Table of Contents

XIO

I don't really know what XIO means — maybe extended input/output? In any event, Atari uses the term to indicate those DOS (and other OS) functions that are possible while in a BASIC environment. The XIO codes are cryptic but useful. They allow you to lock, unlock, delete, rename and otherwise re-specify a file without calling DOS.

XR

DISKEY's variable for the EOR Sector Map mask. This mask allows data to be read with selected bits toggled to compensate some of the simpler encryption techniques.

DISKEY

BY SPARKY STARKS